

# A Scalable Tools Communication Infrastructure

Darius Buntinas  
Mathematics and  
Computer Science Division  
Argonne National Laboratory  
Argonne, IL 60439  
Email: buntinas@mcs.anl.gov

George Bosilca  
Innovative Computing Laboratory  
The University of Tennessee  
Knoxville, TN 37996  
Email: bosilca@eecs.utk.edu

Richard L. Graham  
National Center For  
Computational Sciences  
Oak Ridge National Laboratory  
Oak Ridge, TN 37831  
Email: rlgraham@ornl.gov

Geoffroy Vallée  
Mathematics and  
Computer Science Division  
Oak Ridge National Laboratory  
Oak Ridge, TN 37831  
Email: vallee@ornl.gov

Gregory R. Watson  
High Productivity Computing  
IBM T.J. Watson Research Center  
Hawthorne, NY 10532  
Email: grw@us.ibm.com

**Abstract**—The Scalable Tools Communication Infrastructure (STCI) is an open source collaborative effort intended to provide high-performance, scalable, resilient, and portable communications and process control services for a wide variety of user and system tools. STCI is aimed specifically at tools for ultrascale computing and uses a component architecture to simplify tailoring the infrastructure to a wide range of scenarios. This paper describes STCI’s design philosophy, the various components that will be used to provide an STCI implementation for a range of ultrascale platforms, and a range of tool types. These include tools supporting parallel run-time environments, such as MPI, parallel application correctness tools and performance analysis tools, as well as system monitoring and management tools.

## I. INTRODUCTION

A large number of tools exist to help application developers use high-performance computing systems [1]. Many such tools have been developed as standalone tools, each aimed at solving a specific problem and providing its own run-time support infrastructure, but with little consideration for interoperability with other tools providing complementary capabilities. Such tools tend to be system or architecture specific and hence are more difficult to port and less useful. Some standardization has occurred to help tool developers with their infrastructure requirements. For example, the PAPI project [2] provides a portable interface to system performance data, and the MPI standard [3] provides a communications API. However, little standardization exists for allocating resources, launching parallel jobs, attaching a tool to a parallel job, monitoring, cleanup, and many other services required by tools. Recently [4] [5] it has been recognized that providing a common and portable set of infrastructure

services will be essential for enabling more extensive tool capabilities and new types of analysis tools to be created. Ultrascale systems (petascale and beyond) in particular will place much greater demands on scalability and robustness of these low-level infrastructure services if efficacious tools are to be built for such systems.

Since the terms *tool*, *application*, *run-time*, and *infrastructure* can have a wide variety of meanings, it is important to clearly define what we mean by these terms. We use the term *tool* in a broad sense, to denote anything that aids application developers and users in achieving their goal of using the computer system to obtain their desired data. This covers a wide range of functionality, including tools to help users write their computer codes and analyze their correctness and performance characteristics, tools that manage parallel jobs, and tools that are used to analyze and manage the computer system. We use *application* to mean a user-initiated program that will consume computing (and other) resources in order to achieve a desired outcome. We use the *run-time* to refer to the software component of a computer system that is responsible for launching, monitoring, and terminating a parallel application, as well as managing the low-level communications between application processes. We use the term *infrastructure* to mean those services that are common across a broad set of tools, such as acquiring resources, launching a parallel application, launching a tool, sending data between cooperating tool processes, and even providing graphical user interface primitives.

The STCI collaboration was formed to address tools *infrastructure* needs at the ultrascale. This includes an application programming interface (API) that is independent of system architecture and an implementation design that is guided by ultrascale and multi-tool require-

ments. Neither of these characteristics precludes efficient implementations aimed at the common scale. However, our efforts are focused on the ultrascale. STCI services will provide full parallel job life-cycle management and the communications support infrastructure needed to develop highly scalable tools. We elected to provide a rich set of communication primitives to service a wide range of communications patterns, including an emerging need for multicast/reduction-style communications. The implementation focuses on high performance, scalability, robustness, and portability. Highlights of the infrastructure capabilities include the following:

- A multicast/reduction-style network for scalable communication between a tool user interface and data sources and sinks. This enables the tool to efficiently communicate with large numbers (hundreds of thousands or millions) of independently executing processes and provides a customizable mechanism for data broadcast and aggregation.
- Primitives to support aggregate and point-to-point communication that can be used for efficient startup and intertool communication.
- Scalable system resource management services that enables the tool to obtain and manage the required resources through an abstract interface.
- Tool lifecycle management that will control all aspects of the tool operation, from launch to final shutdown.

The design incorporates a modular component-based architecture to simplify development, deployment, and installation of the infrastructure on a wide variety of ultrascale systems.

The remainder of this paper is organized as follows. Section II presents previous effort for the specification and implementation of scalable run-times for high-performance computing; Section III presents an overview of the STCI architecture; Section IV presents STCI components for the management of the tool and infrastructure lifecycle; Section V presents components for session management; Section VI presents components for communications; Section VII presents components for state persistency; and Section VIII presents components for security. Section IX summarizes the STCI infrastructure.

## II. BACKGROUND

To date there have been a number of other development efforts to provide infrastructure for parallel tool development, although many of these provide support only for a single tool. Many MPI implementations come with their own run-time support, such as LA-MPI's run-time [6], MPICH's Process Management Interface (PMI) [7], and Open MPI's Open Run Time Environment (ORTE) [8]. Some of these attempted to provide general-purpose run-time support; but since these were really

developed in the context of a limited number and type of target tools, they did not meet that goal. A few proprietary implementations of MPI are also available, such as IBM's Parallel Environment [9], and HP-MPI [10], but these rarely provide interfaces that are available for external tools. The Totalview [11] and DDT [12] parallel debuggers provide the infrastructure to deploy debug agents on distributed-memory architectures, then enable these agents to attach to running applications. For MPI jobs, this typically requires support from each MPI implementation (generally known as the Totalview startup mechanism). The infrastructure is also proprietary and not open for other tools to use. The PTP scalable debug manager (SDM) [13] provides an open source framework but relies on external run-time support for launch services. Various parallel performance analysis tools (TAU [14], HPCToolkit [15], etc.) also exist that either use available infrastructure, or provide simple rsh/ssh based launchers for data collection. None of these systems has successfully addressed scalability in ultrascale environments.

Few infrastructures have been designed specifically for third-party tool development and deployment. Notable exceptions are Tool Gear [16] and MRNet [17]. Of these, only MRNet has targeted scalability, but only from the limited perspective of scalable communication between a large number of tool processes. Neither provides any mechanism for tool and application launch, monitoring, or other STCI services.

By addressing the specific issues of interoperability, portability, and scalability of high-performance computing (HPC) run-time support, STCI fills a significant gap that has existed in the HPC software stack for a considerable time.

## III. OVERVIEW

In the ultrascale environment, monolithic tools are no longer a feasible option. Instead, a tool must comprise a number of cooperating parts that collectively achieve the functionality that the tool has been designed for. This *tool model* underpins the architecture that tool designers will need in order to effectively use the STCI services. In this model, the three parts that collectively make up a tool are as follows:

- A *tool front-end*. This is the part of the tool that typically interacts with user, such as a Tool's GUI. A tool front-end interacts with STCI via the *front-end API*.
- One or more *tool agents*. Tool agents provide the means for the tool to interact with the outside world. For example, tool agents are used by a debugger to control application process or by a performance tool to monitor and collect tracing and profiling

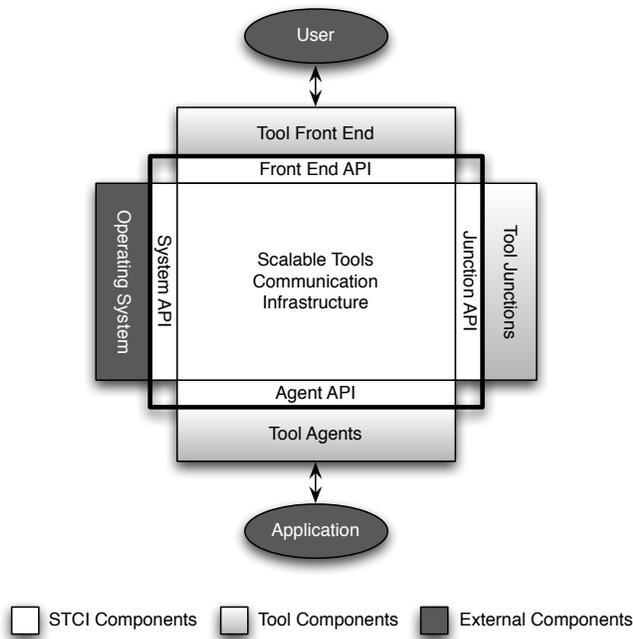


Fig. 1. STCI tool model.

information. The tool agents interact with STCI via the *agent API*.

- One or more *tool junctions*. Tools that must interact with many agents or process large volumes of data use a multicast/reduction-style network [17]. The tool junctions can be used to aggregate, filter, modify, and transform messages sent between the front end and the agents in order to provide an efficient mechanism for dealing with scalability issues.

While technically the tool developer needs to supply only the front end, in an ultrascale environment a tool will typically provide agents and junctions as well. STCI is then used to connect these tool components together. Figure 1 shows how the parts of the tool are related in the model.

The infrastructure provided by STCI is responsible for managing the interactions between the tool parts and the external computing environment via a range of services and activities that are undertaken in response to tool initiated actions, external events, or communication requests.

Tools interact with STCI through a set of APIs that define the services that STCI provides. The APIs are an abstraction the tool developer uses to interact with the target system or application. In addition to this abstraction, STCI provides an implementation that supports a range of different systems and architectures. Since the infrastructure is designed to provide robust and high-

performance services for ultrascale platforms, it includes services that may not be required at more modest scales. In addition, the services needed to help tools scale well on these large systems depend on the underlying hardware support for scalability and reliability. Therefore, the STCI services will be separated into *mandatory* and *optional*. Tools wishing to use optional services will need to verify that these services are available in the implementation they are using.

We use a hierarchical component framework that enables different implementations of component subsystems to be used or additional functionality added (such as support for a new job scheduler or network interface) without requiring changes to any of the existing components. The component architecture employed by STCI will largely follow that used by the Open MPI project [18] and other dynamic component frameworks, such as OSGi [19]. These frameworks typically have the following characteristics:

- A service registry that provides a repository for component descriptions and that supports component publishing and discovery.
- A component interface definition that specifies a contract between the component and other components that wish to employ its services.
- A component life cycle that defines the phases of a component's existence. These phases usually correspond to installation, initialization, activation, deactivation, finalization, and uninstallation.
- A mechanism for resolving dependencies between components.

At the top level of the hierarchy is a logical grouping of components that provide related services. These services include the following:

- **Execution Contexts.** Components in this group are responsible for tool and infrastructure lifecycle management, process execution management, and system resource management. In response to a tool launch request, STCI will undertake the activities required for infrastructure startup/shutdown (bootstrapping), staging tool components, starting/stopping tool execution, and establishing communication between the infrastructure and tool components. When a tool requires resources for operation, STCI will manage the discovery, allocation, and deallocation of system or network resources by interacting with job schedulers, run-time systems or other external resource allocation mechanisms. The execution context components are discussed in more detail in Section IV.
- **Sessions.** All interactions between a tool and STCI are defined in the context of a *session*. A session must be established by the tool front end before

any other activities can take place. STCI sessions are discussed in more detail in Section V.

- **Communications.** Communications are required for many aspects of infrastructure and tool operation. A range of communication primitives for point-to-point, group, and multicast/reduction operations are available for tool implementations. The STCI communication infrastructure is discussed in more detail in Section VI.
- **Persistence.** Components in this group are responsible for managing persistent state information, such as publish/subscribe services, information required for attaching and detaching front ends, and session and tool agent location. Persistence is discussed in more detail in Section VII.
- **Security.** STCI must ensure that the policies and procedures of the local security domain are met. These include activities such as the authentication and management of user credentials and authorization of activities that will be undertaken on behalf of the user. The security components and services are discussed in more detail in Section VIII.

The set of services deployed on a given system is expected to depend on local system policies. For example, some systems allow permanent root-level daemons to run on the system, whereas some don't. Other system characteristics, such as the native process startup mechanism, will affect the types of services that are available.

In the remaining sections, we describe services that will be provided as part of the STCI reference implementation. Component implementation details and interface definitions are beyond the scope of this paper.

#### IV. EXECUTION CONTEXTS

Execution components are responsible for the three phases of running a parallel tool:

- 1) Managing the life cycle of the infrastructure, including the installation and deployment of the STCI services framework
- 2) Interacting with the system resource management services to obtain resources, such as processing elements (which we call locations), required by the tool and associated parallel application
- 3) Managing the life cycle of the tool and application run

STCI provides three sets of components, one for each of the phases of running a parallel tool. These are components for bootstrapping, resource management, and execution context management.

The bootstrapping components are responsible for both system-level infrastructure operation and bootstrapping the infrastructure required for the parallel tools, as these share many common characteristics. They provide

the means to install and uninstall system-level and user-level STCI resources and use these resources for tool startup. In addition, several modes of user-level job startup and control are provided to suit different system requirements:

- Infrastructure installed as a system service, with tool startup occurring using deployed system-level *service agents* (typically these will be daemon processes) and a UNIX-style *setuid* approach to support multiple concurrent users.
- User-level installation of service agents, which are used for tool startup.
- Parallel tool run-time bootstrap of the STCI infrastructure.

On systems that support service agents, these will be deployed to provide services such as standard I/O manipulation and tool-level process monitoring. Depending on the target system architecture, these may also be used to start the tool agents. In some cases a different native mechanism will be required to start the tool agents. These components also provide for an orderly shutdown of STCI management resources, for both expected and abnormal parallel application termination.

The resource management components are responsible for discovering what resources are available for use. They interact with the system resource manager and run-time to find out what resources are available, allocate these resources, and query to see what resources were actually allocated. The instantiation of these steps is system specific, as individual systems may combine some or all of these activities into a single activity, with the STCI infrastructure left to discover what resources have been acquired and which are in use.

The execution context management components manage the execution contexts (or running processes), within the STCI framework. They monitor execution, respond to changes in execution context such as process failure or orderly shutdown, and initiate changes to these execution contexts such as ordered or forceful shutdown. Since application launching is highly system-specific, a range of components will be available to cater to these system differences.

Since the class of tools we are targeting is focused on ultrascale environments, the base design provides the features needed in such extreme computing environments. High performance, scalability, and fault tolerance are key elements in the design. High performance in the context of lifecycle management is addressed by separating bulk allocation of resources at startup from single-event type operations, such as response to failure of a parallel tool's single execution context. Managing the state information obtained at startup, which is needed to respond to run-time tool failures, is largely decoupled from process startup. Scalability of run-time is achieved

by taking advantage of group operations at startup and termination, by localizing run-time monitoring of the running tool, and by providing group communications operations that can be used in process-control operations. Scalable resource utilization is achieved by having sparse network connectivity, conserving the resources needed to manage these resources. Fault tolerance is enabled by providing a publish/subscribe system to help keep track of the state of the system, letting the persistent state components know when an execution context fails, if these have registered for such events. In return, these components inform the execution context components how to respond to this failure, whether to attempt some sort of recovery or to shut down the parallel tool. In addition, redundant connectivity is used in the administrative network, to help survive network failures. In general, a modular design is used so that capabilities, such as support for fault tolerance, can be tailored for specific use case scenarios and can be deployed incrementally.

### V. SESSIONS

All activities undertaken by a tool are carried out in the context of *sessions*. To create a session, the tool must provide an authenticated set of user credentials. Sessions persist until they are no longer needed by the tool. Each session comprises a *resource allocation* (where resources can mean any physical resources required by the tool, such as CPUs or network adapters), a set of tool agents, and a description of how the agents are mapped onto the resources.

Sessions can also contain a number of *streams*. A stream connects the front end to one or more agents, optionally via one or more junctions. Figure 2 shows the structure of a typical stream. The junctions can be used to modify messages passing through them, for example,

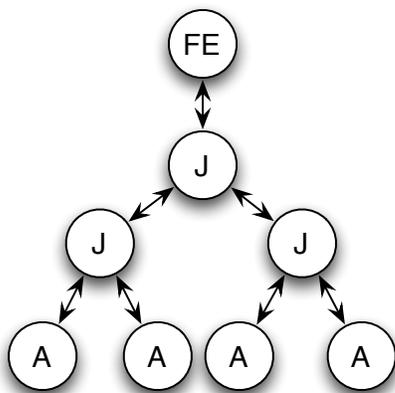


Fig. 2. Stream connecting the front end (FE) to agents (A) via junctions (J).

junctions can aggregate or filter messages sent from agents to the front end or can distribute messages sent from the front end to the agents.

The tool controls how the junctions, agents, and front end are connected by specifying a *topology*. STCI provides predefined topologies (e.g., a binary tree topology) but custom topologies can also be defined by the tool. The topology is then mapped onto a set of resources. STCI can automatically map junctions and agents to available resources; however, it is often useful for the tool to be able to specify where specific agents or junctions are to be located. STCI allows the tool to specify specific resources (e.g., *node3141*) or just a class of resources (e.g., any node with a local disk) where agents or junctions are to be deployed.

Figure 3 shows a topology being mapped onto resources to produce a stream. In the figure,  $j_n$  are the set of tool junctions,  $a_n$  are the set of tool agents, and  $r_n$  are the allocated resources. The stream comprises a connected set of junctions and agents mapped to resources  $j_n r_n$  and  $a_n r_n$  respectively.

The set of junctions, the topology describing the layout of junctions and the mapping of the junctions onto resources are all managed in the context of a session.

### VI. COMMUNICATIONS

Supporting tool communication is the main purpose of STCI. It is therefore critical that STCI provide high-performance, scalable, fault-tolerant communication operations. Furthermore, the communication programming interface needs to be flexible and easy to use. STCI provides an active-message [20] communication programming interface that allows for a high-performance implementation that fits well with the asynchronous com-

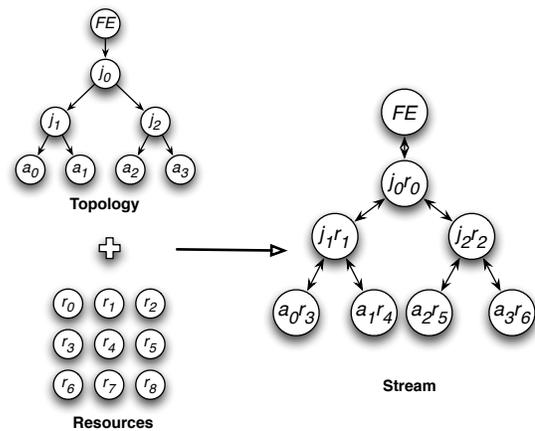


Fig. 3. A stream is constructed by mapping a topology onto a set of resources.

munication patterns of interactive and data aggregation operations performed by tools.

Communication in STCI, whether point-to-point or group communication, is performed over a stream. Streams can support different levels of reliability and ordering guarantees to allow tools to trade reliability and ordering for performance and scalability. A tool will typically have multiple streams, each used to communicate with different subsets of agents or to transfer different classes of data. It is expected that stream communication will be the primary communication method for most tools.

A group communication, such as broadcast, reduction, or allgather, is performed over a stream by implementing junctions to perform the operations. As a convenience, STCI includes built-in *group communication streams* that provide optimized implementations of certain group communication operations.

STCI provides operations to allow complex datatypes to be described efficiently by the tool. These operations allow noncontiguous data to be described so that it can be efficiently communicated. The ability to describe datatypes is necessary to support communication in an environment with heterogeneous architectures. STCI uses the datatype description to convert the binary representations of the basic datatypes from the sender's format to the receiver's format. This conversion operations is performed transparently to the user.

## VII. PERSISTENCE

Persistent state is maintained by STCI to help preserve internal consistent state, to restore the infrastructure to a consistent state, and to provide the facilities for front-ends to disconnect and reconnect to a running parallel tool without negatively impacting this tool. For scalability purposes, the design decouples resource initialization (such as execution context startup) from storing the persistent information relevant to these resources (such as execution context parameters) and monitoring these resources. In addition, in order to deal with unexpected changes in resource availability, a publish/subscribe mechanism is used to track these resources, and notify registered components when changes happen. The persistent state components and the data associated with them are decoupled from the front end, as well as the back-end tool agents, and include a policy service, session management services, event management, and persistent data storage.

Policy service management components direct the response of the parallel tool to state changes, whether this be a front-end agent connecting or disconnecting, an unexpected change in the resources available to a running application, orderly shut-down, or other changes. They are responsible for maintaining consistent policies across

the entire running job. These components implement the policies set by the parallel tool.

Session management services maintain session consistency and provide support for attaching front ends with the session information needed to interact with the running sessions, whether this be for monitoring purposes or for interacting with the running STCI application. These services also manage the information need for merging existing sessions.

Event management components include components used to detect relevant events and are intended to take advantage of monitoring capabilities provided by the system. A publish/subscribe mechanism is designed to inform the subscribed elements asynchronously when relevant events occur.

Persistent data storage components are used to maintain persistent data storage and to allow for storage strategies to be tailored to particular needs, without having to provide new implementations for the components that use this data.

## VIII. SECURITY

The security services provided by STCI are responsible for managing and controlling interaction between users, tools, applications, and system resources according to the policies within a single security domain. This is in contrast to other security mechanisms such as the Distributed Computing Environment (DCE) and Grid Security Infrastructure (GSI), which operate across multiple security domains. The component architecture of the STCI reference implementation allows different components to be used to support multiple policies within a single domain.

The STCI can operate in two security modes: *system mode* and *user mode*. In system mode, the infrastructure is established at system startup and runs with elevated privileges. In user mode, the infrastructure is bootstrapped when the tool (or application) is launched and runs with normal user privileges. An infrastructure running in system mode can provide services to multiple tools, initiated by different users, simultaneously. This necessitates a more stringent security model than that required when running in user mode.

The security services provided by STCI include the following:

- Session authentication. A tool must supply credentials to STCI when a session is first created or when reconnecting to an existing session. Agent-initiated connections to the infrastructure must also be authenticated. The credentials are authenticated by using policies for the local security domain.
- Service authorization. Once authenticated, a tool has access to services for which it has authorization. In particular, this service guarantees that a tool

will not have access to any greater privileges that would normally be available to a particular user. A component that is performing services on behalf of a tool (for example, requesting system resources), must use this service to ensure that the action is authorized.

The STCI security model has been deliberately kept as simple as possible to avoid conflicting with any existing security mechanisms. In addition, STCI does not provide any services for data encryption or data integrity.

## IX. CONCLUSION

The development of efficient and scalable tools for high-performance computing (HPC) has always been a challenge. With the emergence of petascale platforms and the next-generation exascale systems, it is now becoming critical. The lack of a standard for HPC tool infrastructure has resulted in myriad implementations, but few have addressed the scalability and portability demands that these new systems will place on tool developers.

This paper presents the architecture of the Scalable Tools Communication Infrastructure (STCI), which aims to provide a simple, modular, and standard infrastructure for the implementation of scalable run-time systems and tools for high-performance computing, especially targeting peta- and exascale platforms.

STCI provides a standard API that tool developers can use to ensure their tools remain portable and scalable across a wide range of architectures. The infrastructure is composed of components for scalable communications and process control services that address five main areas: *execution contexts*, *communications*, *sessions*, *persistence*, and *security*. These basic services provide all needed mechanisms and policies for the implementation of scalable tools, the management of communications in a scalable manner, and the management of user session and security issues.

Using this architecture, STCI provides a unifying platform for the implementation of a wide variety of tools for high-performance computing. These include support for scalable debuggers, run-time support for parallel programming libraries such as MPI, and support for scalable operating system tools such as fault-tolerance frameworks.

## ACKNOWLEDGEMENTS

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

## REFERENCES

- [1] M. Collette, B. Corey, and J. Johnson, "High performance tools and technologies," *Lawrence Livermore National Laboratory Tech. Report UCRL-TR-209289*, December 2004.
- [2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.
- [3] Message Passing Interface Forum, "MPI: A message passing interface standard," <http://www.mpi-forum.org>, June 1995.
- [4] Eclipse.org, "PTP Developers Workshop," [http://wiki.eclipse.org/PTP/workshops/May\\_2007](http://wiki.eclipse.org/PTP/workshops/May_2007).
- [5] Oak Ridge National Laboratory, "Software Development Tools for Petascale Computing Workshop," <http://www.csm.ornl.gov/workshops/Petascale07/>.
- [6] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski, "A network-failure-tolerant message-passing system for terascale clusters," *International Journal of Parallel Programming*, vol. 31, no. 4, pp. 285–303, August 2003.
- [7] Argonne National Laboratory, "MPICH2," <http://www.mcs.anl.gov/mpi/mpich2>.
- [8] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg, "The open run-time environment (OpenRTE): A transparent multi-cluster environment for high-performance computing," in *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.
- [9] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea, "The communication software and parallel environment of the IBM SP2," *IBM Syst. J.*, vol. 34, no. 2, pp. 205–221, 1995.
- [10] Hewlett Packard Corporation, "HP-MPI," <http://www.hp.com/go/mpi>.
- [11] Etnus, LLC, "TotalView," <http://www.etnus.com/TotalView>.
- [12] Allinea, "Alinea DDT – A revolution in debugging," <http://www.allinea.com/DDT.pdf>.
- [13] G. R. Watson, "A Model-based Framework for the Integration of Parallel Tools," in *Proceedings of the 2006 IEEE International Conference on Cluster Computing*. IEEE Computer Society, September 2004.
- [14] S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, 2006.
- [15] J. Mellor-Crummey, "HPCToolkit: Multi-platform tools for profile-based performance analysis," Invited presentation at the 5th International Workshop on Automatic Performance Analysis (APART), Phoenix, AZ, November 2003.
- [16] John Gyllenhaal and John May, "Tool Gear," [https://computation.llnl.gov/casc/tool\\_gear](https://computation.llnl.gov/casc/tool_gear).
- [17] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proceedings of SC2003 (SC'03)*, Phoenix, AZ, November 2003.
- [18] J. M. Squyres and A. Lumsdaine, "The component architecture of Open MPI: Enabling third-party collective algorithms," in *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds. St. Malo, France: Springer, July 2004, pp. 167–185.
- [19] OSGi Alliance, <http://www.osgi.org>.
- [20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th International Symposium on Computer Architecture*, June 1992, pp. 256–266.